Internet-in-a-Box: Emulating Datacenter Network Architectures using FPGAs
Jonathan Ellithorpe

Research Project
Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of Master of Science, Plan II.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Randy Katz
Research Advisor

(Date)

********

Professor David Patterson
Second Reader

(Date)
# Table of Contents

Table of Contents .................................................................................................................. 2

List of Tables and Figures ......................................................................................................... 3

Abstract .................................................................................................................................... 5

1 Introduction ............................................................................................................................. 5

2 Emulation Methodology .......................................................................................................... 6

  2.1 A Set of Composable Models ............................................................................................ 7

  2.2 Functional and Timing Split ............................................................................................. 8

    2.2.1 Example .................................................................................................................... 8

    2.2.2 Design Rational ......................................................................................................... 9

  2.3 Virtualization ...................................................................................................................... 9

3 Emulation Architecture .......................................................................................................... 11

  3.1 Node Model ...................................................................................................................... 11

    3.1.1 Processor Model ....................................................................................................... 12

    3.1.2 Network Interface Controller Model ....................................................................... 13

    3.1.3 Timer Model ............................................................................................................ 15

  3.2 Switch Model ..................................................................................................................... 17

  3.3 Scheduler .......................................................................................................................... 17

  3.4 Logging ............................................................................................................................. 18

  3.5 Software ........................................................................................................................... 19

4 Example - Putting It All Together ........................................................................................ 20

5 Evaluation .............................................................................................................................. 21

  5.1 Resource Consumption ..................................................................................................... 21

  5.2 Performance ..................................................................................................................... 22

  5.3 Switch Model Validation - TCP Incast ........................................................................... 24
List of Tables and Figures

Figure 1. We model the datacenter as a set of switches, routers, links, and servers. We develop composable hardware models for each of these key components and use them as building blocks to construct a model of a datacenter network, an example of which is shown on the right. Additionally, we let a centralized scheduler synchronize the models. ................................................................. 7

Figure 2. Mapping an Example Target into a Functional and Timing Model. .............................................. 8

Figure 3. Example of Model Virtualization. Left - Two switches map to two separate models. Functional models and timing models each have only a single context. Right -- Here we have a single switch model with virtualized hardware emulating two contexts. The figure shows that switch 1 is currently getting to run on the hardware, while switch 0 waits to be scheduled. ................................................................. 10

Figure 4. High Level Architecture Overview ................................................................................... 11

Figure 5. Processor Timing Model and its Interaction with the Processor ............................................. 12

Figure 6. 64-bit IIAB Packet Descriptor .................................................................................. 14

Figure 7. NIC Model ........................................................................................................... 14

Figure 8. Hardware Timer Model .............................................................................................. 16

Figure 9. Switch Model ........................................................................................................... 17

Figure 10. Illustration of Synchronization Barriers and Relative Emulation Progress of Models ........ 18

Figure 11. IO Address Space. ................................................................................................... 20
Figure 12. Mapping a target system into our emulation architecture. (a) The target is a 64-node datacenter rack with a 64-port output buffered layer-2 switch. (b) Diagram describing the logical setup and componentry of our emulation system. (c) Mapping the logical emulation setup (b) into our emulation architecture.

Figure 13. TCP Window Illustration.

Figure 14. (a) Scenario we wish to simulate. (b) Our simulation setup. The switch model sends and receives packets to and from a TCP implementation in C using DPI. The switch itself is simulated using MentorGraphics ModelSim 6.5a.

Figure 15. Node 0 TCP SeqNo Trace.

Figure 16. Goodput versus senders for 1MB and 256KB output buffers in the switch.

Figure 17. 8 Senders, 256KB Queues. Illustrates the relationship between the senders' windows and queued packets. (a) Packets in the receiver's queue over time. Red dots indicate drops. (b) Window sizes for each sender over time. They experience a synchronous timeout. After 350ms all senders cyclically timeout for the rest of simulation, shown in (c).

Figure 18. (a) Close-up of the receiver's queue size over the course of 40ms. (b) Nodes' window sizes over the same time interval.

Figure 19. Global Chip Synchronization.

Figure 20. (Top) Models think they communicate directly with each other. (Bottom) In reality, models are talking to a communicator, which abstracts away the inter-chip communication.

Table 1. Internet-in-a-Box System Calls

Table 2. FPGA resource consumption on an LX110T

Table 3. Performance of simulating / emulating the Internet-in-a-Box target processor, running a floating-point dot-product workload.

Table 4. Simulation Parameters

Equation 1. Emulation Throughput

Equation 2. Emulation Slowdown Factor of the Switch Model.
Abstract
In this report I describe the design and evaluation of Internet-in-a-Box, an FPGA-based research tool for exploring network architectures in large datacenters (i.e., on a scale of $O(10,000)$ servers). Our approach to emulation involves constructing a model of the target architecture by composing simplified hardware models of key datacenter building blocks. In our current implementation, those building blocks are the datacenter’s switches, routers, links, and servers. We implement models in programmable hardware, which gives designers full control over emulated buffer sizes, line rates, topologies, and many other network properties. Such control also gives researchers complete observability into the network. Additionally, because our node model emulates servers using a full SPARC v8 ISA compatible processor, each node in the network is capable of running real applications. We can then flexibly compose models together to emulate any number of network architectures. Although each model emulates its target component separately, a centralized scheduler is responsible for synchronization. We break each model down into a "functional model" and a "timing model", and we show the benefits of using this approach. We also virtualized each model in time, increasing overall emulation throughput (as defined later by Equation 1). We show that by using these two techniques we are able to emulate 256 nodes and 5 switches on a single Xilinx Virtex 5 LX110T FPGA, and argue that we will be able to scale to $O(10,000)$ nodes in the future using the BEE3[7] system. Finally, we evaluate our design and validate our switch model by reproducing and studying the TCP Incast throughput collapse problem in datacenters [8].

1 Introduction
In recent years, datacenters have been growing rapidly to scales of up to 100,000 servers [1]. Many key technologies make such incredible scaling possible, including modularized datacenter construction and server virtualization. The change in scale enabled by these advances draws great attention to datacenter networking architecture. To explore this large design space, researchers typically use software simulation on virtualized servers, which can only reach a scale of $O(10)$~$O(100)$ nodes [2][3]. Additionally, such approaches often fail to represent the true timing characteristics of the proposed architectures, thus leading to less creditable results. Recently, however, cloud computing vendors have offered pay-per-use services to enable users to share their datacenter infrastructure at an $O(1,000)$ node scale, e.g., Amazon EC2, Microsoft Azure and Google App Engine [4][5][6]. Such services, though, provide almost no visibility into the network. Furthermore, these platforms have no mechanism for experimenting with new networking elements or devices. To address the above limitations, we propose
an inexpensive and reconfigurable emulation testbed for datacenter network research, called "Internet-in-a-Box". We show how it is possible to scale our system to $O(10,000)$ nodes, which is a size that matches that of medium to large sized datacenters today. Furthermore, because we can run real applications on our node models, we can explore complex network behavior that administrators see when deploying equivalently scaled datacenter software. Instead of fully implementing the target system, we build several models targeting key datacenter components and compose them together in hardware. With the latest Xilinx Virtex 5 FPGA, we can build a system capable of emulating 256 nodes on a single chip at over 100MHz. We can then construct a 10,000-node system from a rack of multi-FPGA boards (e.g., the BEE3 [7] system). The performance is slower than a real implementation, but is still several orders of magnitude faster than event-driven software simulators. We perform logging at the hardware level, giving researchers clear visibility into the network.

The rest of this report is structured as follows: Section 2 discusses some terminology and high-level emulation concepts, including our notion of "models", timing models, functional models, and hardware virtualization. Section 3 describes our hardware architecture component by component, and Section 4 presents an example of how to map a target system into Internet-in-a-Box, featuring a full system architectural diagram. In Section 5, we validate the switch model using a TCP simulation of the Incast throughput collapse problem studied in [8], and in Sections 6 and 7, we evaluate our architecture and discuss limitations and future work.

2 Emulation Methodology

In this section, we describe our conceptual approach to emulating datacenter network architectures. First, we explain our breakdown of a datacenter into a set of models and the advantages of this modeling technique. Second, we dig deeper into how we have chosen to represent each model in hardware and explain our design rational. Finally, we discuss hardware virtualization and how it allows us to increase emulation throughput (see Equation 1 for definition of emulation throughput).
2.1 A Set of Composable Models

FPGAs and similar programmable ASICs have been widely used in the field of scalable system design verification, such as Cadence Palladium [13] and MentorGraphics Veloce [14]. These multi-million dollar boxes allow designers to implement a fully functional model of the target chip and run it at a frequency of around 1 to 2 MHz. We define the Target here to mean the system we want to emulate, while the Host refers to its physical representation on FPGAs. Unlike those systems, however, Internet-in-a-Box is designed to emulate a much larger target system (i.e., a datacenter), and therefore we do not implement a fully functional target. Neither do we implement a complex monolithic model of the target. Rather, we have developed simplified hardware-based models of key datacenter building blocks, including switches, routers, links, and servers (see Figure 1). Additionally, we have made these models composable by developing standardized communication interfaces between them. Lastly, each model "outsources" the responsibility of synchronizing itself with other models to an external scheduler. This approach offers the designer several powerful advantages:

1. Complex models of new datacenter architectures can be composed quickly.
2. New models can be designed in isolation.
3. Bugs and design errors are easier to locate.
4. Centralized scheduling makes experimenting with synchronization policies easy.
5. Declarative languages can be used to describe architectures in the future.
6. Scaling to multiple FPGAs is straightforward. We postpone a discussion of this topic until the section on Future Work.
In summary, providing the user a set of datacenter building blocks increases the effectiveness of our system as a research tool.

2.2 Functional and Timing Split

Each of the models described in the previous section are composed of two separate pieces: a functional model and a timing model. We implement both types of models in hardware. Functional models are simply the transistors necessary to implement the desired functionality of the target, independently of timing or performance. Timing models track the progression of time passed in the target (typically counted in cycles). They track time by 1) observing what operations the functional model is completing now, 2) knowing how much time T in the target would have passed in order to complete those operations, and 3) incrementing a time tracking counter by T. We provide the following detailed example.

2.2.1 Example

Say that our target is a magic in-order single-issue MIPS processor with perfect caches, bypassing, and prediction, and therefore has a CPI = 1 for any workload.
It would be impossible to implement such a processor on an FPGA, so we choose our functional model to be something much simpler and consumes far less hardware, yet can do everything the magic MIPS processor can do. That is, our functional model is a MIPS processor with no caches, no bypassing, no prediction, and so little control logic that every instruction travels through the pipeline alone to avoid any hazards. The resulting pipeline has a constant CPI of 5. Figure 2 illustrates this scenario.

Instructions flow through the simplified pipeline over the course of several host cycles (FPGA cycles). When an instruction commits, the timing model knows that in the target, such an operation would take only one cycle, so the timing model increments time by one. After this point, if an outside observer (e.g., the scheduler) were to ask the timing model for the time, the timing model would reply "1" and would continue to do so until the next instruction committed.

2.2.2 Design Rational
There are other approaches to consider for hardware modeling. Primarily, implementing a fully functional target (similar to some processor emulation tools available), or developing a monolithic model of the target. Compared to these approaches, however, explicitly separating the functional and timing models has the following key advantages:

1. Reduced hardware costs on FPGAs compared to implementing a fully functional target model. Because the functional model alone lacks any notion of timing, we can neglect any performance optimizations in the target (see example above). The result is that models consume less hardware and we can fit more of them on an FPGA.
2. Timing models are interchangeable, allowing a designer to change the system characteristics like link speeds and delays without modifying the functional model.
3. The timing model only encodes the timing characteristics of the target, and can be as simple as a set of counters. We show later that this has been our experience by describing our resulting emulation architecture.

2.3 Virtualization
One design goal of Internet-in-a-Box is to maximize emulation throughput per FPGA, defined in Equation 1.
Equation 1. Emulation Throughput

\[ \text{Emulation throughput} = \left( \frac{\text{models}}{\text{FPGA}} \right) \times \left( \frac{\text{Target cycles}}{\text{Host cycle model}} \right) \]

That is, emulation throughput is the sum of the average number of target cycles that each model on an FPGA can complete in a host (FPGA) cycle. One way of increasing emulation throughput is to replicate the hardware models to fill up the chip (increase the models per FPGA term). Previous work has shown that within the domain of processor emulation, this technique is suboptimal [9]. In fact, by "virtualizing" a host processor the overall emulation throughput in MIPS/FPGA can be twice that of using the naïve model replication method. By "virtualization" of a model we mean the use of multiple hardware contexts (a complete set of target state variables) sharing the model's core data-path in time (Figure 3 shows an example). The concept is exactly analogous to "hardware multithreading" in modern processors. The increase in emulation throughput because of virtualization is due to its ability to hide emulation latencies and increase memory bandwidth utilization. That is, while the first term in Equation 1 increases linearly with the number of hardware contexts used, the second term decreases sub-linearly, resulting in an overall increase in the product. Therefore, we introduce the idea of virtualization to our own hardware models to improve the emulation throughput of our system.

Figure 3. Example of Model Virtualization. Left - Two switches map to two separate models. Functional models and timing models each have only a single context. Right -- Here we have a single switch model with virtualized hardware emulating two contexts. The figure shows that switch 1 is currently getting to run on the hardware, while switch 0 waits to be scheduled.
Figure 3 illustrates an example of how we use this concept in the switch model. Note that we virtualize both the functional and the timing model together, and each timing model context is responsible for tracking the emulation progress of its corresponding functional model context (e.g., buffering and forwarding packets).

Our current result is that we are able to reach an emulation capacity of 256 nodes and 5 switches on a single Xilinx Virtex 5 LX110T FPGA, as shown in Section 5.1.

3 Emulation Architecture

In this section, we describe our emulation architecture. Figure 4 shows the higher-level pieces, including a node model and a switch model. The node model targets a networked computer and talks over Ethernet to a switch targeted by the switch model. The scheduler is simply responsible for synchronizing the models to maintain emulation accuracy. After walking through the architecture in detail, we will give an example of how to use it to model a target system.

3.1 Node Model

Our node model targets a datacenter server, and is composed of a processor, a hardware timer, and a Network Interface Controller (NIC). Figure 4 illustrates this subdivision. Each of these components is further broken down into their functional and timing components, and each are virtualized in time to act as 64 separate instances (through the use of 64 hardware contexts). The result is that a single node model in our architecture is capable of emulating a maximum of 64 independent target servers. The
functional models encapsulate the hardware necessary to perform all relevant functions of the target, while the timing models encode how these operations map into time.

3.1.1 Processor Model

The processor functional model implements the 32-bit SPARC v8 ISA, and is a single-issue in-order pipeline that runs at over 100MHz. We verified the model against the certification test suite donated by SPARC International. We can therefore run unmodified SPARC binaries. A single functional model pipeline is virtualized using *host multithreading*, which refers to the usage of N hardware threads (or contexts - we use the terms interchangeably) in the host sharing the pipeline in time. We have chosen N to be 64 for FPGA hardware mapping efficiency. This is because the Virtex 5 FPGA platform is equipped with 6-input LUTs, some of which can also be used as memories, making 64 a natural fit. Hence, we can use a single pipeline to emulate up to 64 separate processors in the target. We can emulate more by replicating the processor model.

![Diagram of Processor Timing Model and its Interaction with the Processor](image-url)
Each hardware thread also shares access to a 32-bit IO Bus, allowing each thread to communicate with its corresponding context in the NIC Model or Hardware Timer Model. We accomplish thread identification on the IO Bus using a thread ID (tid) signal. Each thread can therefore have its own independent view of the IO devices to which it has access. In our current implementation, the Hardware Timer and NIC are the only IO devices, and each support 64 hardware contexts.

Figure 5 shows the timing model architecture, which tracks the number of completed target cycles for each target processor. The timing model has the following control mechanism over the functional model: on every cycle, the timing model tells the CPU whether to inject the next instruction for thread TID, using its tid_out and valid_out signals. We service each thread in round-robin order for simplicity. If the timing model injects a valid instruction, then some cycles later at the end of the pipeline the functional model will tell the timing model whether the instruction is completing (given by the retired signal). If an instruction is retiring, then the timing model will increment the target cycle count in its memory for that thread. The count will be incremented by the number of cycles it takes to complete that instruction in the target. For instance, if a floating-point divide is completing, the timing model may encode that such an operation takes 12 cycles, and increment the thread's target cycle count by 12. The model of instruction timing encoded in our system, at present, is simply that each instruction takes one target cycle to complete. Since instructions do not always retire in their first pass through the pipeline (e.g., load and store instructions that miss in the cache), each thread will progress at a slightly different pace depending on its workload. When a thread has completed a number of target cycles equal to LOCAL_SYNC_BARRIER, we stop injecting instructions into the pipeline for that particular thread while we allow other threads to proceed. We increment the "Barrier Hit Count" when the control logic finds that a thread has reached its barrier. We know that all threads are finished when the counter reaches 64. At this point, the timing model sends its barrier_hit signal to the scheduler. The proceed signal from the scheduler lets us know that it is ok to continue, and resets the Barrier Hit Count.

3.1.2 Network Interface Controller Model
The second major component of the node model is a NIC, which we virtualized to support one for every thread. Our NIC model sends and receives 64-bit layer-2 packet descriptors in place of real packets. The limits of FPGA resources motivate our use of descriptors (less than 1MB of on-chip memory for an LX110T FPGA). Figure 6 illustrates the fields. We have reserved the TAG portion for looking up packet payloads in main memory, although we have not currently implemented payload functionality. The
receiving node, for example, may use this field to transfer the payload into his memory space, or a network router may use it to inspect the packet payload.

| SRC MAC | DST MAC | LENGTH | TAG |

**Figure 6. 64-bit IIAB Packet Descriptor**

Figure 7 shows the NIC functional model, which buffers packet descriptors from the processor model for transport into the network. The NIC observes the particular context being addressed from the IO Bus (via the tid signal), and places the incoming packet into the TX Queue for that context.

**Figure 7. NIC Model**

Similarly, packets destined for node X from the network are stored in node X's receive queue via the switch model interface (a standard data/valid signaling interface). Because packets are 64-bits and the IO bus is 32-bits, a small half-packet memory is used on the processor side to temporarily store the first half of the packet before it is sent to the transmit queue. The NIC will send an interrupt signal to the processor if the RX Queue contains valid packets for the thread currently controlling the IO Bus. The NIC is only one of potentially many IO devices supporting interrupts for thread TID, and so we connect this interrupt signal to an interrupt controller. The controller observes all interrupt requests in the given
cycle and generates the appropriate signal to the processor. We can disable interrupts for the NIC if a polling style is preferred.

Because we have separated the functional and timing models and virtualized them in time, it is not necessary for us to implement 64 independent queues. Instead, we have implemented each set of 64 queues using a single-port memory and access each queue one at a time. The result is reduced hardware consumption. We let the timing model control the completion of target cycles to reflect the parallel and independent nature of the queues in the target system.

Figure 7 also shows the NIC timing model, which determines at what target cycle the packet at the head of each TX Queue should move to the next network element. It does this by using its Packet Delay Counters, which count down the number of target cycles for packet transmission. The timing model therefore, has the ability to control the link speed for each NIC. We do this by using the "Delay Calculator" module, which decodes the length field and translates it into the number of target cycles required to transmit a packet of that size. For example, if the NIC's timing model specifies a 1Gbps link and 1ns target cycles, a 64B packet translates into a 512-cycle delay in the target. We process queues in round-robin order using the Queue Select counter. When the count value read from the memory for the queue indicated by "Queue Select" is equal to one, then the timing model tells the functional model to transmit the packet, and the count is decremented. If the count value is zero, the size of the corresponding queue is checked. If there exists another packet then its transmission delay is calculated and loaded into the counter. When the timing model finishes servicing all queues, it increments the target cycle counter to indicate that a target cycle has passed for each target NIC. When this count reaches LOCAL_SYNC_BARRIER, all operations are halted until the proceed signal allows us to continue by resetting the target cycle counter.

3.1.3 Timer Model
The last component of the node model is a hardware timer used for creating a notion of timing for software running on the processor, like TCP. Like all other models, we have divided the timer into a functional model and a timing model. The architecture is quite similar to the models described so far, and we include a description of it here for completeness.
The functional part is composed of a memory to hold the current timer values for each thread and some control logic to interpret the commands of the IO Bus and the timing model. In any given cycle, the IO Bus may write a new value, in microseconds, to the timer memory. If the thread context on the IO Bus has its timer value equal to one then we trigger an interrupt and send it to the interrupt controller for processing. We clear the interrupt by writing a value of zero to the timer register for that thread.

Since the tid signal from the IO Bus cycles in round-robin order indefinitely, the timing model also uses it to index into the timer memory. Because the timers are on the granularity of microseconds, the timing model must know when a microsecond of time has passed before decrementing their values. It does this by incrementing the Microsecond Counter after completing each target cycle, and waiting for it to equal the number of target cycles per microsecond, given by the CYCLES_PER_US emulation parameter. For example, if we are emulating 1GHz CPUs, then a target cycle is 1ns and the Microsecond Counter will count to 1000 before the timing model can decrement the timers. The Target Cycle Counter is also incremented on every target cycle until we hit the LOCAL_SYNC_BARRIER, at which point the timing model tells the scheduler that it has reached the barrier and halts any further progress. The proceed signal resets the counter and frees the model to continue emulating the target.

Figure 8. Hardware Timer Model
3.2 Switch Model

Our switch model targets a 64-port, 1Gbps, output buffered layer-2 switch, and has a very similar architecture to that of the NIC model. The functional model is composed of 64 queues implemented in a single-port memory, and forwarding logic to direct incoming packets to the correct output queue. We size the queues to model, in approximation, the size of the target's packet buffers. We say approximate because the switch model stores a fixed number of packets, not bytes. The timing model controls the transmission of these packets across a single on-chip link that we have multiplexed in time to emulate 64 physical links in the target. In our current implementation, the switch stores 64 packets per output buffer using on-chip SRAM. We have the freedom to size these buffers to model the target's buffering capacity, up to the limits of FPGA resources. Beyond this point, we can turn to using external DRAM as well. In the case of using DRAM, the timing model can slow the passage of target cycles for the switch in the event of a high-latency access.

3.3 Scheduler

The scheduler is responsible for observing all the signals from the timing models that indicate whether they have hit their synchronization points. Figure 10 shows an example of what the progress of threads might look like at a particular snapshot in time. Once all models have reached their barriers, then the
scheduler tells the models to proceed. Naturally, if the barriers are set farther apart then we can achieve higher performance by amortizing the overhead of synchronization. On the other hand, we decrease emulation accuracy as a result. Because our goal in using Internet-in-a-Box is to run very large-scale experiments in some reasonable amount of time, we will tune this parameter to give us the maximum accuracy while still allowing the experiment to be tractable. The exact effects of tuning this parameter have yet to be characterized, although initial experiments with the processor alone show that 60 host MIPS (not emulated MIPS) can be achieved even when LOCAL_SYNC_BARRIER is set to one (see Table 3), its lowest possible value.

![LOCAL_SYNC_BARRIER Diagram](image)

Figure 10. Illustration of Synchronization Barriers and Relative Emulation Progress of Models.

### 3.4 Logging

Although not shown in Figure 4, the logger taps interesting signals in the system and generates log records for sending to off-chip storage. Our current implementation of the logger records packet events in the switch (receiving, transmitting, and dropping) and stores these records in a FIFO. We placed the FIFO at a special address on the IO Bus where a thread is capable of reading out the records. This thread then sends them to an on-chip Ethernet controller for delivery to a computer. We have left as future work the design a more sophisticated log delivery mechanism. Our current implementation, however, supports 6MB/s of logging bandwidth and 512 log entries of buffering capacity, sufficient for observing a single switch.
3.5 Software

To get the entire system up and running, we write a code image to main memory and instruct the processor to start fetching instructions. We accomplish this using two components: the application server, and the Ethernet controller. The application server runs on a nearby computer and streams the code image to the Ethernet controller, which DMAs the image into memory. When finished, the Ethernet controller sends a special signal to the processor telling it to begin execution. Each hardware thread will then start fetching instructions from its own portion of memory. We segmented main memory 64 ways so that each thread has 32MB of storage on a 2GB DIMM.

The code image itself consists of two parts: the application and a small kernel. The kernel starts at address zero and first bootstraps the processor, readying it for application code execution. When done, the kernel hands control over to the application.

Since the application code runs in user mode, it must access the hardware through system calls. It does this by issuing a trap instruction for the system call handler entry in the trap table, with the correct system call index number. Table 1 shows the system calls supported.

<table>
<thead>
<tr>
<th>System Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int sys_iowritew( int* ioaddr, int data )</code></td>
<td>Write word to IO address</td>
</tr>
<tr>
<td><code>int sys_iowritel( int* ioaddr, int data_msw, int data_lsw )</code></td>
<td>Write double word to IO address</td>
</tr>
<tr>
<td><code>int sys_ioreadw( int* ioaddr, int* data )</code></td>
<td>Read word from IO address</td>
</tr>
<tr>
<td><code>int sys_ioreadl( int* ioaddr, int* data_msw, int* data_lsw )</code></td>
<td>Read double word from IO address</td>
</tr>
<tr>
<td><code>void sys_register_callback( interrupt_callback func, int callback_id )</code></td>
<td>Sets the user-level interrupt handler for the NIC and timer using a callback</td>
</tr>
</tbody>
</table>

The trap table also contains entries for handling various levels of interrupts. Registered in this table, therefore, are two interrupt handlers, one for the hardware timer, and one for the NIC. When we call these handlers, they in turn jump to the user-level handler registered using the `sys_register_callback` function. The NIC handler is responsible for draining the receive buffer, while the timer handler should clear the timer. Figure 11 shows the IO address space for these operations. Queuing a packet involves only writing a double word to the TX Queue memory. Similarly, we de-queue a packet by reading a double word from the RX Queue. We control the timer by the same protocol.
4 Example - Putting It All Together

Our target in this example is a datacenter rack filled with 48 single CPU (1GHz) HP server blades and an HP ProCurve 2900-48G 1Gbps 48-port top-of-rack switch. Although we do not know the allocation algorithm for buffer space to incoming packet streams in the switch, we assume here for simplicity an output buffer mechanism, with equal capacity per port. We also assume that each blade has only one NIC. We first translate the target system into abstract emulation components as in (b), and then into a detailed emulation architecture in (c). In this example, our abstract emulation setup consists of 48 node models, a switch model, a scheduler, and a logger. We emulate 48 nodes by virtualizing the node model hardware (processor and NIC model in this example). Packets flow from the processor to the NIC's transmit queue before we inject them into the network. Forwarding logic in the switch decides in which output queue to store the incoming packets, and we size the queues to match the capacity of the target. For example, the ProCurve 2900-48G switch has ~23MB of buffering [10], which translates to 23,000 1KB packets. This also implies 480 packets per output buffer. If the queue is full then we immediately drop...
the packet. The timing model then monitors each of the queues in the switch’s functional model, and maps packet transmit events into simulation time as a function of the packet sizes and link speeds. Since the CPU is operating at 1GHz, we conveniently chose a target cycle to be 1ns to match the cycle time of the processor. Thus, the switch’s timing model will take 8,192 target cycles to transmit a 1KB packet. We note here, however, that we can define the granularity of simulation time (i.e., the definition of a target cycle) differently for every model in the system. It is the responsibility of the scheduler to interpret the progress of each model and synchronize them at the right points in time. In any case, the transmitted packet will flow to its destination's RX queue in the NIC model, and interrupt the destination's thread of execution. The interrupt handler will call the registered user-level handler, where the application logic may take appropriate action.

During all of this, the timing models track the completion of target cycles and synchronize at each LOCAL_SYNC_BARRIER. If we choose to do synchronization at the microsecond granularity, for example, then LOCAL_SYNC_BARRIER is equal to O(1,000) target cycles.

5 Evaluation

In this section, we analyze the architecture, discuss its performance, point out bottlenecks, and use the system to study the TCP Incast problem.

5.1 Resource Consumption

Table 2 outlines the resource consumption for our system on a Virtex5 LX110T FPGA. Given these resource requirements, we can instantiate 4 processor pipelines, 4 NICs, and 5 switches each with a 7,680 packet buffering capacity on an LX110T FPGA. Since each packet descriptor represents a real packet of size between 64B and 1500B, 7,680 packets translates to between 0.5MB and 11MB of emulated capacity per switch. If more capacity is required to emulate the target, then we can use DRAM and the timing model can patch up the high latency accesses in simulation time.

A single $2,000 FPGA platform [11] is therefore capable of emulating a 256-node system with a network of 5 switches. It is clear by looking at Table 2 that our limiting resource is on-chip memory. Thus, projecting forward to the largest 40nm Xilinx Virtex6 FPGA (SX475T), we will have 6x more resources to either spend on more models or increase the complexity of our models. In terms of packet buffering capacity in the network, that is approximately 500,000 supported in-flight packets.
### Table 2. FPGA resource consumption on an LX110T

<table>
<thead>
<tr>
<th></th>
<th>Register Bits</th>
<th>LUTs</th>
<th>BRAM (18kb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>4886 (7.1%)</td>
<td>2839 (4.1%)</td>
<td>30 (10.1%)</td>
</tr>
<tr>
<td>Switch</td>
<td>53 (0.1%)</td>
<td>137 (0.2%)</td>
<td>30 (10.1%)</td>
</tr>
<tr>
<td>NIC</td>
<td>77 (0.1%)</td>
<td>250 (0.4%)</td>
<td>4 (1.4%)</td>
</tr>
<tr>
<td>Memory Controller</td>
<td>2472 (3.6%)</td>
<td>1975 (2.9%)</td>
<td>10 (3.4%)</td>
</tr>
</tbody>
</table>

### 5.2 Performance

Our emulation system runs slower than the target system in terms of wall clock time. There are two contributing factors to this slowdown: a) FPGA clock speeds, and b) virtualization. Since the FPGA runs at ~100MHz, emulating a 1GHz target system contributes 10x to our slowdown. In addition, because we virtualize our hardware models into 64 time slices, there is an additional performance penalty of 64x. However, host multithreading on the processor can hide emulation latencies and offset the impact. For example, we see in Table 3 that at 100MHz the processor model can emulate target processors at 60 MIPS, which is close to 1 MIPS per target processor. Therefore, if the target is a 1GHz processor with a CPI of 2, we have a slowdown factor of approximately 530. Table 3 also shows that our hardware emulation techniques beat software simulators using the latest computer hardware by a factor of ~7. We note here that the timing model used in these experiments synchronized each thread in lock-step, incurring the highest possible synchronization overhead. Furthermore, the processor model at the time of writing this report is capable of running at over 130 MHz.

All other models in the Internet-in-a-Box system take 64 host cycles to complete a single target cycle for their respective target systems. For example, in the switch timing model the control logic must check and decrement each of 64 counters in the single-ported packet delay counter memory. Therefore, the performance of the switch model relative to the target switch is a simple function of the FPGA frequency and the granularity of time we wish to emulate:

\[
\text{Equation 2. Emulation Slowdown Factor of the Switch Model.}
\]

\[
\text{Slowdown}(\text{HOST}_\text{frequency}, \text{TARGET}_\text{timeunit}) = \frac{64}{\text{HOST}_\text{frequency} \times \text{TARGET}_\text{timeunit}}
\]
The above expression gives us the number of wall clock seconds it takes to emulate a single second of target time. For example, if the FPGA is running at 130MHz and the granularity of time we wish to emulate is 10ns, then we can calculate the slowdown to be:

\[
\frac{64}{(130 \frac{cycles}{microsecond}) \times (0.010 \text{ microseconds per timestep})} = 49 \text{ seconds per target second}
\]

In these calculations, 64 represents the performance cost of virtualization. For example if we were to model a 32-port switch, it would take us only \(\sim 25\) seconds to emulate a second in the target system. We note here that the processor and the other models in the system do not have to agree on the value of a target time unit (or target cycle, as we have referred to it previously). The timing models and the scheduler are together responsible for tracking simulation time and synchronizing models. Models like the processor are execution driven and may require a high degree of accuracy because the target can change its state (e.g., execute an instruction) as fast as every nanosecond. The switch, on the other hand, is an event-based model where the events are packet transmissions. If we emulate a 1Gbps switch and define the minimum packet size to be 64B, then packet transmissions can occur as fast as every 512 nanoseconds. Therefore, the switch may define its target time unit to be on the order of 10s of nanoseconds and still achieve a high degree of accuracy with respect to event orderings and packet delivery times. We can therefore achieve higher accuracy with finer granularity time steps at the cost of emulation performance. Another good example is the timer model, which counts in microseconds. Therefore, it is natural to define a target cycle for the timer to be a microsecond.

Table 3. Performance of simulating / emulating the Internet-in-a-Box target processor, running a floating-point dot-product workload.

<table>
<thead>
<tr>
<th>Simulation Method</th>
<th>Run time for simulating the functional model</th>
<th>Run time for simulating both functional and timing model</th>
<th>Host MIPS for simulating both functional and timing model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware emulation on FPGA (100 MHz)</td>
<td>N.A.</td>
<td>17.8 sec</td>
<td>60.4 MIPS</td>
</tr>
<tr>
<td>Software simulation on Nehalem (3 GHz)</td>
<td>46.5 sec</td>
<td>121.9 sec</td>
<td>8.8 MIPS</td>
</tr>
<tr>
<td>Software simulation on Barcelona (2.3 GHz)</td>
<td>58.7 sec</td>
<td>192.5 sec</td>
<td>5.6 MIPS</td>
</tr>
</tbody>
</table>
5.3 Switch Model Validation - TCP Incast

The "TCP Incast Problem" refers to the collapse in goodput (i.e., application level throughput) experienced when a single client accesses data from multiple servers across a datacenter network [8]. Such a situation is common when data are striped across many storage nodes. First, we provide a brief overview of TCP [12].

5.3.1 TCP Overview

TCP provides end-to-end reliable and in-order delivery over unreliable networks by using a sliding window technique. TCP tags each packet with a sequence number that indicates its order in the stream of transmitted information (see Figure 13). The receiver gets these sequence numbers, potentially out of order or unreliably, and responds by sending ACKs back to the sender indicating the highest sequence number that it received in order (via a sequence number in the ACK itself). If the ACKs received by the sender indicate that the receiver got more data (the ACK sequence number is greater than the sender's Max ACKd variable), then two things happen: 1) the TCP sender moves its left window edge to the highest "un-ACKed" piece of data, and 2) potentially increases its window size. After increasing the window size and sliding it forward, the sender transmits unsent data and records the greatest sequence number of data that it sent. This describes the "normal" operation of TCP. When a packet is lost or reordered in the network, then the receiver will get sequence numbers out of order and simply ACK for the next packet that it needs in sequence, for every received out-of-order packet. The sender then will receive "duplicate" ACKs. That is, ACKs with the same sequence number. If three duplicate ACKs are received in a row, the sender assumes that a simple re-ordering did not occur, but rather that a packet loss has occurred. This is because receiving three packets out of order is an improbable situation in most networks that TCP is used. TCP responds to three duplicate ACKs by halving its window size and resending the missing packet. We refer to decreasing the window size in this manner as multiplicative decrease, while the normal behavior when receiving regular in-series ACKs is to additively increase the window size. Hence, we call TCP's congestion avoidance scheme "additive increase, multiplicative decrease". There is an exception to the multiplicative decrease, however, which is when the sender times out because it has been too long since they last received a packet from the sender. At this point, they set their window size to one and enter "slow start". Slow start is the exception to additive increase, and in fact is where the window increases exponentially until it hits a certain threshold. The purpose of slow start is to let the sender ramp up quickly at startup or after a timeout to get back up to speed. In the next section, we use our switch model to reproduce experimental findings in real-world
environments of poor TCP performance when multiple senders are transmitting to one receiver over a one-hop network.

![TCP Window Illustration](image)

Figure 13. TCP Window Illustration.

![Simulation Setup Diagram](image)

(a) Scenario we wish to simulate. (b) Our simulation setup. The switch model sends and receives packets to and from a TCP implementation in C using DPI. The switch itself is simulated using MentorGraphics ModelSim 6.5a

5.3.2 Experimental Setup

To verify our switch model design we attempted to reproduce the TCP Incast throughput collapse over a one-hop network. Figure 14(a) illustrates our target system, Figure 14(b) shows our simulation environment, and Table 4 shows our simulation parameters. We simulate the behavior of the nodes in
the network using a C implementation of TCP, which communicates over DPI with the switch model RTL, simulated in ModelSim. Our TCP model implements the basic functionality, including slow start, additive increase and multiplicative decrease of window sizes, and fast retransmissions. The TCP senders transmit 1KB packets continuously to node 63, the receiver, for about 18 seconds of simulation time. The granularity of simulation time steps is one microsecond. We varied both the number of senders between 1 and 63, and experiment with queue sizes of 256KB and 1MB.

<table>
<thead>
<tr>
<th>Switch Parameters</th>
<th>Value</th>
<th>TCP Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet Size</td>
<td>1024B</td>
<td>Duplicate ACK Threshold</td>
<td>3</td>
</tr>
<tr>
<td>Random Start Time (sender)</td>
<td>0 ms – 10ms</td>
<td>Slow Start</td>
<td>enabled</td>
</tr>
<tr>
<td>Link Bandwidth</td>
<td>100Mbps</td>
<td>Fast Retransmission</td>
<td>enabled</td>
</tr>
</tbody>
</table>

### 5.3.3 Results

Figure 15 shows an example set of sequence numbers sent and ACK numbers received for a single sender in a group of 63 sending to a single receiver across our switch. The switch in this case is a 64-port output buffered switch, with 1MB of buffering capacity per port.

![Figure 15. Node 0 TCP SeqNo Trace.](image-url)
The sender in Figure 15 experiences an initial RTT of approximately 3ms, followed by 7ms, 15ms, 26ms, 40ms, and finally 63ms, however the node times out at 150ms before it can receive the ACKs for the last set of packets sent. The exponential nature of the RTTs is attributable to the exponential growth of the 62 other senders' window sizes. Since each sender sees the same view of the network, all nodes experience the same timeout at approximately the same time.

Figure 16 shows our goodput results for various configurations of senders and buffer sizes, and show that for 1MB buffers we have been able to reproduce the general trend of falling goodput as we increase the number of senders. Interesting to note is the increase in goodput when using 256KB packet buffers for a large number of senders. Although these data points merit further investigation, we hypothesize that smaller queues effect smaller round trip times and therefore faster timeouts in TCP, allowing TCP to discover and recover from loss faster. Prior work has shown that decreasing the time for timeouts can indeed increase goodput [8], strengthening our suspicion that this may be the case.

Figure 16. Goodput versus senders for 1MB and 256KB output buffers in the switch.

Figure 17 shows the queue size over time for an experiment with eight senders and 256KB packet queues in the switch, along with a graph of each node’s CWND over time for the first 550ms of simulation.
Figure 17. 8 Senders, 256KB Queues. Illustrates the relationship between the senders' windows and queued packets. (a) Packets in the receiver's queue over time. Red dots indicate drops. (b) Window sizes for each sender over time. They experience a synchronous timeout. After 350ms all senders cyclically timeout for the rest of simulation, shown in (c).
Just before the timeout at ~320ms, the average window size is about 33 packets, implying a total of more than 256KB of data in flight. It is at this point that we see the queue beginning to overflow, and all nodes timeout at about the same time as a result. During this timeout period the receiver's queue drains, and eventually the senders retransmit the lost packets. The receiver gets the lost packets, and responds that it has in fact received quite a bit of data after it. The senders respond by rapidly opening their windows, resulting in very high packet loss and a second synchronous timeout. From that point forward no sender ever increases its window beyond 16 for the rest of simulation, as shown in Figure 17(c). Figure 17(a) shows that during this time the receiver's queue is always close to full, and drops packets at a very regular rate (calculated to be 76 packets per second on average for the rest of simulation). Figure 18 shows a close-up view of this pattern alongside the window sizes for each sender, illustrating the correlation between senders opening their windows and successive peaks in the queue size graph.
6 Future Work

Our future work primarily involves developing the system's functionality and scaling across many FPGAs.

6.1 Sending Payloads, Not Just Descriptors

The current design does not support sending packet payloads, which will be necessary when running real applications on the system. We propose here two possible implementations of such a mechanism. Our first idea is to use reserved "network storage" space in DRAM to store all packet payloads that are in flight. A controller who maps tags from the IIAB packet TAG field to memory locations could manage such storage. If network elements drop packets, then the responsible model would let the controller know to free the memory set aside for storing that payload. Upon delivery of a packet to a node, the controller may take control and DMA the packet into the receiving node's storage set aside for the NIC. Alternatively, the receiving node may take control instead, and request a packet transfer from "network storage" to a specified address in his memory segment. In the latter case, it would be necessary to ensure that each receiving node responsibly does this so that packets do not overflow the network storage space over time. Another idea for moving payloads is to not actually move them at all, and simply transfer the notion of a memory page from one node to another, fixing the virtual address translations as necessary. The system would then notify receiving nodes that a packet has simply entered their memory and has the following address. This mechanism has the potential to have low overhead and high performance, since we are only changing a few pointers in the system to move the packet.

6.2 Creating a Large System and Scaling to Multiple FPGAs

Our discussion so far in this report has been limited to focusing on a single FPGA. To scale to multiple FPGAs we must develop communication and synchronization mechanisms that work across chips.

Our current idea for inter-chip scheduling extends the synchronization mechanisms already in place. That is, we allow on-chip scheduling to continue as usual, except that the local schedulers report to a global scheduler. The global scheduler simply waits for all local schedulers to report that their on-chip models have finished simulating some number of synchronization intervals. Figure 19 illustrates this idea, showing that we may set our global synchronization interval in terms of LOCAL_SYNC_BARRIERS, given by the GLOBAL_SYNC_BARRIER parameter.
To allow our models to communicate across FPGAs, we propose two high level mechanisms. The top of Figure 20 shows what the communicating models think is going on, while the bottom reveals that a communicator module in fact tricks each model into thinking that they are talking directly. The communicators are responsible for packetizing the communications and sending these packets to an inter-chip communication engine (ICCE). The ICCE is aware of the inter-chip communication mechanisms available (e.g., on the BEE3 system) as well as where each model resides, allowing it to determine the correct destination and forward the packet appropriately.
7 Conclusion

In this report, we have described the Internet-in-a-Box datacenter network emulator and system architecture. As currently implemented, the design consists of a node model and a network model synchronized by an on-chip scheduler. The node model is broken down into separate models for its processor, network interface controller, and hardware timer. We split all models into their functional and timing components for boosting emulation efficiency and flexibility, and we describe each model's architecture in detail. Given these emulation components, we have demonstrated how to map a rack of 64 machines with a 64-port rack switch into our emulation architecture. We have shown our progress towards validating our switch model by using TCP workloads to reproduce the TCP Incast throughput collapse effect. We also were able to observe the switch at the architectural level and easily see the relationship between application level behavior (TCP window sizes) and hardware architectural phenomena (buffers overflowing in the switch).

Finally, we have shown that our system is capable of emulating 256 nodes connected in a network of 5 switches, also on a single FPGA system costing $2,000. The relative cost of ownership for such a system compared to a 256-node cluster makes our platform attractive for researchers. Moreover, every researcher can own one on their desktop as a private research testbed, as opposed to time-sharing hundreds of machines in a large machine room.

8 Acknowledgements

Parts of this report were taken from a paper co-authored with Zhangxi Tan and Randy Katz submitted to the Design Automation Conference 2009. I would like to thank them both for being incredible mentors and providing invaluable insight throughout the design process. I would also like to thank my colleagues Andrew Waterman, Yunsup Lee, and Rimas Avizienes for their help in getting the processor model working and using the related RAMP infrastructure.

9 References


